LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-410283

Architectural Implications for Spatial Object Association Algorithms

V. S. Kumar, T. Kurc, J. Saltz, G. Abdulla, S. R. Kohn, C. Matarazzo

February 2, 2009

IEEE International Parallel & Distributed Processing Symposium
Rome, Italy
May 25, 2009 through May 29, 2009

Disclaimer

Architectural Implications for Spatial Object Association Algorithms*

Vijay S. Kumar
Department of Computer Science and Engineering, The Ohio State University

Tahsin Kurc, Joel Saltz
Center for Comprehensive Informatics, Emory University

Ghaleb Abdulla, Scott R. Kohn, Celeste Matarazzo
Center for Applied Scientific Computing, Lawrence Livermore National Laboratory

Abstract

Spatial object association, also referred to as crossmatch of spatial datasets, is the problem of identifying and comparing objects in two or more datasets based on their positions in a common spatial coordinate system. In this work, we evaluate two crossmatch algorithms that are used for astronomical sky surveys, on the following database system architecture configurations: (1) Netezza Performance Server®, a parallel database system with active disk style processing capabilities, (2) MySQL Cluster, a high-throughput network database system, and (3) a hybrid configuration consisting of a collection of independent database system instances with data replication support. Our evaluation provides insights about how architectural characteristics of these systems affect the performance of the spatial crossmatch algorithms. We conducted our study using real use-case scenarios borrowed from a large-scale astronomy application known as the Large Synoptic Survey Telescope (LSST).

1 Introduction

Object association, also known as cross-correlation or crossmatch, is the process of identifying and comparing objects or entities present in different datasets. Datasets in applications involving object association are often acquired from multiple surveys or observations of objects using instruments, e.g., observation of objects in the sky by telescopes in an astronomy application. Each object has a unique identifier and a set of qualitative and quantitative features including its spatial coordinates. Object data are generally maintained in lists or catalogs. These catalogs are incrementally updated with newer observations of the same or different objects over time and hence the sizes of these catalogs may grow in time.

The crossmatch problem can be defined as follows: Given two or more datasets of objects, the goal is to find for *each and every* object in one dataset, all objects in the other datasets that lie within a certain "distance" from the object. Here, the notion of distance is defined based on an application-specific metric. When two objects are compared based on their positions in a common spatial domain, then the problem is referred to as the *spatial object association* or *spatial crossmatch*. The distance metric generally corresponds to the Euclidean distance or the angular separation between the objects in the domain. The spatial crossmatch problem is commonly encountered in many application domains. For example, a biomedical researcher may wish to match observations from multiple microscopy images to study the temporal evolution of cancerous cells within an organism. As another example, in astronomy, astronomers seek to crossmatch celestial objects extracted from images of sky regions that are captured over time at different wavelengths and under different experimental conditions. In this case, objects observed by a telescope in a given time period may need to be compared and matched with objects in a historical catalog constructed from earlier observations.

The catalog-based nature of object association naturally lends itself to the use of relational database systems for data management. In this paper, we investigate and evaluate a set of database systems with different architectural configurations and identify per-

*This research was supported in part by the National Science Foundation under Grants #CNS-0403342, #CNS-0426241, #ANI-0330612, #CNS-0203846, #ACI-0130437, #CNS-0615155, #CNS-0406386, and Ohio Board of Regents BRTTC #BRTT02-0003 and AGMT TECH-04049, the NIH U54 CA113001 and R01 LM009239 grants, and the NHLBI R24 HL085343 grant.

formance differences and bottlenecks in these systems for spatial crossmatch use-cases mainly driven by the astronomy community. We focus on spatial crossmatch in the context of a large-scale astronomy application known as the Large Synoptic Survey Telescope (LSST) [1]. Our experiments are based on real use-case scenarios put forth by the LSST astrophysicists. We investigate the execution of two well-known, database-oriented crossmatch algorithms using 1) Netezza Performance Server®, a parallel database management system with active disk style architecture support for certain types of database operations, 2) MySQL Cluster, a database system designed for high-availability and high-throughput, and 3) a distributed collection of independent database system instances with support for data replication.

2 Motivating Application

We investigate spatial crossmatch in the context of a large-scale astronomy application known as the Large Synoptic Survey Telescope or LSST [1]. The LSST is a wide-field survey reflecting telescope. When it becomes operational, it will photograph the entire sky every three nights. The LSST presents hitherto unprecedented data acquisition rates, thanks to a 3.2 Gigapixel camera that can capture an image of a sky region – corresponding to the field-of-view (FOV) of the telescope – every 15 seconds. This will translate to about 15-20 terabytes of data acquired each night. All image data will be archived and is expected to top out at 55 petabytes after 10 years [5]. The LSST catalogs are expected to contain around 50 billion objects at the end of the survey.

One of the principle scientific goals of the LSST project is to enable the detection of small objects and transient events that occur deep in the solar system. Astrophysicists have put together several analysis pipelines to realise this goal. One of the pipelines is the *association pipeline* (AP), which crossmatches the new detections for each sky region against objects in a historical catalog with a user-specified search radius. A common search query in this analysis pipeline is the n-way crossmatch query: "*Given n catalogs of objects, for each object belonging to one catalog, determine all potential matching objects from the remaining $n-1$ catalogs*", i.e. determine all objects in the other catalogs that lie within a search radius of d arcseconds from the source object. In database parlance, this crossmatch query corresponds to a *spatial join* operation, i.e., a join between the object catalog table (henceforth referred to as `Object`) and the object detections tables (henceforth referred to as `DIASource` because the new detections are obtained using Differential Image Analysis performed on light sources) based on the spatial coordinates and other spatial attributes of the objects. The crossmatch query helps multiple collaborating astronomers in the community to correlate their observations which are likely to have been gathered at different wavelengths from different geographical locations using instruments with different detection capabilities.

The LSST association pipeline is divided into three phases: a *prepare* phase that extracts and prefetches relevant field of view (FOV) data from `Object` into a new table (`FOVObject`), a *compare-and-update* phase, which performs the distance-based crossmatch between the prefetched data and detections and decides if alerts need to be generated, and a *postprocessing* phase that commits any changes as a result of the crossmatch to disk. Object density within an FOV is estimated to be around 4 million on an average and around 10 million in the worst case. It is also estimated that a snapshot would contain one new detection for every hundred objects within an FOV. Therefore, the association pipeline would involve, at the least, a crossmatch of 40,000 new detections against 4 million objects. The biggest challenge from a computational perspective is the *near real-time* transient alert generation [4]; if a new detection cannot be matched against any known historical object, an "alert" must be triggered in near real time (within 30 seconds of the image capture) so that the astronomy community can closely monitor this previously unknown transient object. Existing database-oriented solutions on uniprocessor configurations are unable to meet the LSST requirements. Our work sought to help the LSST determine the most suitable infrastructure to deploy for this time-critical activity. Hence, we investigate the use of database system configrations over shared-nothing architectures in an effort to improve upon the existing solutions. In section 3, we discuss some related efforts that target the crossmatch challenge.

3 Related Work

Most efforts towards performance optimization of the spatial crossmatch problem for large datasets focus on reducing the computation cost (i.e., minimizing the number of object comparisons) and the disk I/O overheads. We classify these efforts into two broad categories: *customized* solutions and *database-oriented* solutions.

Customized solutions are those in which the crossmatch logic is implemented outside of the database. These solutions are usually application-specific and involve the use of special data structures and specific op-

timizations for a given architecture. For the LSST application, Serge Monkewitz of the Infrared Processing and Analysis Center (IPAC) has developed a shared-memory based solution that performs real-time crossmatch on in-memory object data. Spatial indexes can be employed to speed up the extraction of relevant FOV objects from the `Object` catalog. Over the years, many spatial indexing schemes have been developed for storage and retrieval of spatial data. Gaede and Gunther [7] provide a comprehensive survey of such schemes. Variants of the R-tree index have been used to support nearest-neighbor and other spatial queries in astronomy. The Hierarchical Triangular Mesh (HTM) spatial index by Kunszt [13] recursively divides the sky area into multiple spherical triangles and numbers them based on spatial proximity. Taylor [14] has proposed $\mathcal{O}(N \log N)$ tile-based and multicone indexing approaches for the crossmatch. Papadomanolakis et al. [10] developed an indexing algorithm known as Directed Local Search (DLS) for efficient query processing in unstructured tetrhedral meshes. Gray et al [8] propose a finer-grain "zones" indexing scheme that minimizes computational cost by reducing the search space for each object. While the use of indexes improves data selection times, complex indexing schemes can slow down updates to `Object`. Data compression and incremental clustering are some of the techniques that have been used in the past to reduce disk I/O volume and facilitate efficient data updates.

Crossmatch logic in database-oriented solutions is implemented in the database itself using `SQL`. Popular database systems support a basic set of native spatial indexing schemes such as the R-tree. The more complex indexing schemes may be implemented as stored procedures within the database system but at the cost of performance. Power [11] has experimentally evaluated the performance of crossmatch algorithms on large catalogs using the `MySQL` and `ORACLE` database systems. Szalay et al. [9] have developed an algorithm for the crossmatch that uses a "zones" indexing scheme. They have implemented this algorithm on the Microsoft `SQL Server` system. Becla et al. [5] describe the spatially partitioned storage of the object catalog in the form of sub-tables, where each sub-table contains object data for a region or chunk of the sky. These sub-tables are striped across multiple disks for fast access. No indices are maintained for the sub-tables. As a result, update performance improves at the affordable cost of increased data selection time. Our work is similar to these earlier works in that we target database-oriented solutions to support spatial object association. We study the impact of different parallel architectural configurations and their idiosyncrasies on astronomical crossmatch algorithms and suggest techniques to improve performance on these configurations based on our evaluations.

4 Crossmatch algorithms

In this section, we describe two crossmatch algorithms that are well-known to the astronomy community. We use these algorithms in our experimental evaluation.

4.1 Zones algorithm

The *Zones* algorithm for the astronomy crossmatch was proposed by Gray et al. [8] and makes use of a spatial indexing scheme known as the zones index. The sky is viewed as a sphere referenced using declination and right ascension coordinates that wrap around near the poles. The zones index bins this sphere data horizontally into non-overlapping "zones" or bands of some predefined height such that objects with similar declination values lie within the same zone. Given such a partitioning, the search space within which to look for potential object matches for a detection can now be restricted to the containing zone and a small subset of its "neighboring" zones, collectively known as its neighbor set. This is because the motion characteristics (e.g., orbital paths) of celestial objects are not arbitrary. Figure 1 shows how the spherical sky is partitioned into zones, and how the zones index can be used to reduce the crossmatch search space. For example, in the figure, the zone height and search radius chosen result for crossmatch of an object result in a neighbor set consisting of 3 zones. Further constraints based on the declincation and right ascension ranges within these zones help reduce the search space of objects to the dotted rectangle that borders the search radius.

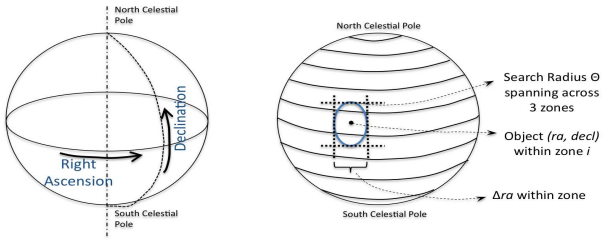

Figure 1. Spatial parameterization of the sky and the use of the zones index

Gray et al. [8] have expressed their crossmatch algorithm in the form of an `SQL` query as shown in Figure 2.

Here, ZoneNeighbor is a precomputed table that maintains the neighbor set information for each zone, *ra* and *decl* are respectively the right ascension and declination coordinates for the object in the sky while x, y and z are its projected spatial coordiantes that enable fine-grain distance-based comparison. We refer the reader to the paper by Gray et al. [8] for a detailed description of the query.

```
SELECT d.objID, o.objID
FROM DIASource d, FOVObject o
INNER JOIN ZoneNeighbor zn
      ON d.zoneID=zn.zoneID
INNER JOIN o ON zn.matchZoneID=o.zoneID
WHERE o.ra BETWEEN d.ra-zn.Δra AND d.ra+zn.Δra
      AND d.decl BETWEEN d.decl-θ AND d.decl+θ
      AND POW(d.x-o.x,2)+POW(d.y-o.y,2)
          +POW(d.z-o.z,2)< d_max
```

Figure 2. SQL query for the *Zones* crossmatch algorithm (DIASource vs. FOVObject)

4.2 Optimized Zones Algorithm

The Optimized Zones (*OptZones*) algorithm proposed by Becla et al. [5] is an improved form of the *Zones* algorithm that exploits the LSST-specific assumption that each zone's neighbor set contains a maximum of three zones: the containing zone itself and the two "sandwiching" zones (i.e., $zone_{i-1}$ and $zone_{i+1}$ for a given $zone_i$) along the declination dimension. The main advantage here is that the neighbor set of a zone can be easily computed on the fly and zone neighbor information need not be precomputed and explicitly maintained like in *Zones*. The algorithm traverses all zones within an FOV in non-decreasing order starting from the least zone ID to the largest zone ID within that FOV. The spatial attributes of all objects that lie within the neighbor zones of each such zone i are loaded into a $\texttt{SecondaryZone}_i$ table created on-demand for zone i. The algorithm then does an all-to-all comparison between detections that lie within zone i and all entries in the $\texttt{SecondaryZone}_i$ table to obtain the crossmatch results for objects in zone i. The same steps are repeated for every zone in the FOV [1]. Becla et al. have implemented the *OptZones* algorithm using SQL and stored procedures supported by certain database systems. Figure 3 shows the SQL equivalent of part of the algorithm that does crossmatch for objects within a zone i.

```
SELECT d.objID, s.objID
FROM DIASource d
INNER JOIN SecondaryZone_i AS sz
      ON d.ra BETWEEN sz.ra-Δra AND sz.ra+Δra
WHERE d.zoneID = i
      AND sz.decl BETWEEN d.decl-θ AND d.decl+θ
      AND POW(d.x-sz.x,2)+POW(d.y-sz.y,2)
          +POW(d.z-sz.z,2)< d_max
```

Figure 3. SQL crossmatch query for zone i in the *OptZones* algorithm

5 Architectures and Database Configurations

Database system configurations on uniprocessor architectures are generally unable to meet the real-time requirements of the LSST crossmatch. In this section, we present three alternative database system configurations running on shared-nothing parallel architectures. These configurations differ with respect to their data storage and retrieval mechanisms, query execution and storage engines.

5.1 Configuration 1: Parallel Database System with Active Disks

The first configuration consists of a parallel database system which runs on a parallel backend and employs active disk style [3] hardware acceleration for some of the database operations. We chose this configuration to evaluate the efficency of parallel query execution (how suitable are the crossmatch algorithms to parallelization?) and to study the impact of active disk style processing on the crossmatch. We used a highly-scalable commercial data warehouse appliance, the **Netezza Performance Server®** (NPS) [2] as a representative for this configuration. The NPS system consists of one frontend and a large number of backend "snippet" processing units (SPUs) all interconnected via a Gigabit Ethernet switch. The frontend does query parsing, parallel query plan generation (i.e. all queries including complex queries like joins are broken down into sub-queries or snippets that are executed in parallel on each SPU) and combines snippet results obtained from all SPUs. Each SPU is an embedded processor equipped with its own memory and

[1] As the zones are traversed in increasing order, the SecondaryZone tables are not created afresh for each zone. Instead, they are refreshed in a 3-way sliding window fashion as we iterate from one zone to the next.

disk. The tables in the database are partitioned and distributed across the disks on the SPUs. NPS does not support the concept of in-memory tables and supports limited caching in the SPU memories; hence, whenever a new query references a table, there is I/O involved as the table's contents need to be retrieved off disk each time. To minimize the I/O, this configuration leverages hardware acceleration provided by active disks. Active disk based systems push computation closer to the data (by offloading processing to disk-resident processing units), as opposed to the conventional idea of staging disk-resident data into memory for processing [3] (Processing units are integrated with disk controllers so that application-specific code executing on these units can process the data at the rate at which it streams off disk). NPS employs active disk style processing by delegating simple database operations such as row-based filtering and column-based projection to pre-programmed Field Programmable Gate Array (FPGA) units placed near the disks of each SPU. In addition to these salient features of Configuration 1, NPS also combines the use of the following additional features:

Hash-based data partitioning: NPS uses hash-based data partitioning to uniformly distribute the rows in each table across all SPUs. In general, such a uniform distribution of data based on a hash of the contents of one or more "distribution columns" will lead to better load balance amongst the SPUs during query execution and hence better performance.

User-defined functions: NPS provides support for application-specific user-defined functions (UDFs) which cannot be expressed using SQL. UDFs are coded in a high-level language and are translated into object code executed in parallel by each SPU on its local data.

Zone-maps: NPS does **not** support index construction on the data. Instead, Netezza provides zone-maps (not to be confused with the *Zones* algorithm) implemented in the software. While indices generally tell a database system what data to read, zone-maps tell the system what not to read. For each new query, a zone-map look-up tells the FPGAs on each SPU what disk blocks not to read.

5.2 Configuration 2: High-Throughput Network Databases

The second configuration consists of a high-throughput network database system which runs on a cluster of shared-nothing high-end processors, each having a large amount (a few Gigabytes) of local memory. Such configurations are not used to optimize performance of a single complex query inasmuch as they are used to support a large number of concurrent simple queries. The motivations behind evaluating such a configuration are two-fold: (1) This configuration uses the collective memory of all cluster nodes to store data; that is, all tables and index data are stored in memory on these nodes and the database system provides the frontend with a virtual unified view of a single, large memory pool spanning all nodes. With increasing affordability of memory and the rise in popularity of non-volatile flash memory, it may be feasible to store entire astronomy catalogs in such memory pools. High-speed interconnects in today's architecures provide rapid access to remote memory in such pools. Using this configuration, the prepare phase and the crossmatch can be performed entirely on in-memory data, thereby leading to significant I/O savings. (2) This configuration is designed to support the execution of multiple concurrent queries. We wish to know if crossmatch algorithms can exploit this feature for improved performance.

We used the open-source `MySQL Cluster` [12] as the representative for this configuration. An instance of `MySQL Cluster` comprises one manager node and a set of data (backend) nodes and API (frontend) nodes. `MySQL Cluster` has a memory-based transactional storage engine that partitions all table and index data and distributes them uniformly (based on a hash of one or more distribution columns) across the memories of the data nodes. `MySQL Cluster` is also a high-availability database, where data can be replicated synchronously so that there is no single point of failure. Data is periodically logged to local disk on each data node to prevent data loss. A query is parsed by the frontend node that it was submitted to. The locally generated query plan is then broadcast to all the data nodes. `MySQL Cluster` does not have a parallel query engine and hence, does not implicitly support parallel execution of all query types. Index lookup and table scans can occur in parallel on all data nodes, making `MySQL Cluster` efficient for point queries and data lookup operations. However, joins and complex queries cannot be executed in parallel on the data nodes. However, unlike NPS, `MySQL Cluster` supports a wide range of distributed indexing mechanisms.

5.3 Configuration 3: Independent Databases with Replication

The third configuration is a hybrid configuration that we designed by borrowing the best features of the earlier two configurations. This configuration consists of a distributed collection of independent database system instances which run on a cluster of shared-nothing high-end processors with large amounts of memory

(i.e., on each such data node, we have a single, independent database server instance). The frontend consists of a single master node responsible for parsing potentially multiple concurrent queries and for sub-query generation, i.e., dividing each query into a set of small sub-queries that can be submitted as a batch for concurrent execution on the data nodes. This configuration does not implicitly support any parallel query plan generation as was the case in configuration 1. Instead, it tries to explicitly exploit coarse-grain parallelism at the sub-query level. Effectively, we have created an opportunistic coarse-grain parallel query engine that seeks to generate small, disjoint sub-queries for a given query. Like configuration 2, this configuration supports index construction and maintains all table and index data in memory on the data nodes. Each table is explicitly partitioned by the master node which then distributes its rows uniformly among the data nodes. However, unlike the earlier configurations, the frontend does not have a virtual unified view of the data on all data nodes because we are dealing with an independent collection of database system instances, each of which have access only to local data. Like configuration 1, joins and complex sub-queries can be executed in parallel on the backend data nodes.

This configuration also supports various degrees of data replication in order to tune performance based on the query workload. In one extreme form of this configuration, data replication is disabled, in which case the data is uniformly partitioned across the data nodes. Here, the querying mechanism will resemble configuration 2, where the master node needs to broadcast every sub-query to all the data nodes. This is because each data node contains a portion of the table referenced by the sub-query; hence, all data nodes are involved in the execution of any sub-query. This is the `DataPartition-QueryBroadcast` execution strategy and is useful for light query workloads. At the other extreme, complete data replication is enabled, i.e., all data is present on all the data nodes. Here, the querying mechanism mirrors configuration 1 in that each sub-query is executed by any one data node. This is the `DataBroadcast-QueryPartition` strategy and is useful for small datasets and heavy query workloads. In general, any intermediate replication strategy can be chosen that will result in data being replicated on and each sub-query being executed by a subset of the data nodes. The novelty of this hybrid configuration is that, by choosing an appropriate replication strategy, we can tune for performance, given any `DIASource` and `FOVObject` sizes.

We used open-source `MySQL` as the database system instance on each data node. The master node (frontend) was implemented in C++ and executes outside of the database. The master node controls the partitioning and replication of data and schedules the execution of each batch of sub-queries among the data nodes. Our implementation of this configuration uses DataCutter [6] for runtime support and parallel execution in a shared-nothing cluster environment. DataCutter is a component-based middleware framework that uses the filter-stream programming model. Application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through a *stream* abstraction. In this configuration, we used a version of DataCutter which employs the MVAPICH flavor of MPI for communication. This enabled us to leverage Infiniband support on clusters (where available) for high-speed communication and data exchange between the `MasterNode` filter and each `DataNode` filter.

Figure 6 summarizes the pros and cons of each of the architectural configurations discussed in this section.

6 Performance Evaluation

In this section, we describe the dataset and query region characteristics used in our experiments and present results from our evaluation of the two crossmatch algorithms under each configuration. In our experimental evaluation, for comparison purposes, we also include as our baseline, a naive approach to crossmatch which takes each detection in `DIASource` and compares it against every object in `FOVObject`.

6.1 Dataset and Query Characteristics

We use the USNO-B catalog (a public astronomy catalog generated by US Naval Observatory at Flagstaff containing over a billion objects) as it closely emulates the characteristics of the data that LSST would generate when it becomes operational. Each object record is around 100 bytes. The crossmatch search radius, θ was set to 3 arcseconds. We use three different test FOV regions to evaluate crossmatch performance. These regions are characterized by differences in their object density (high, average, low). Detections for an FOV region are obtained by applying a perturbation function on the objects within that FOV. This function generates a new detection for roughly every hundred objects in an FOV. Table 1 summarizes the characteristics of each test region including the number of matches that result from crossmatching detections and objects within that region. Our focus is to improve performance for the high density FOV, because of its closeness in density to the average LSST case.

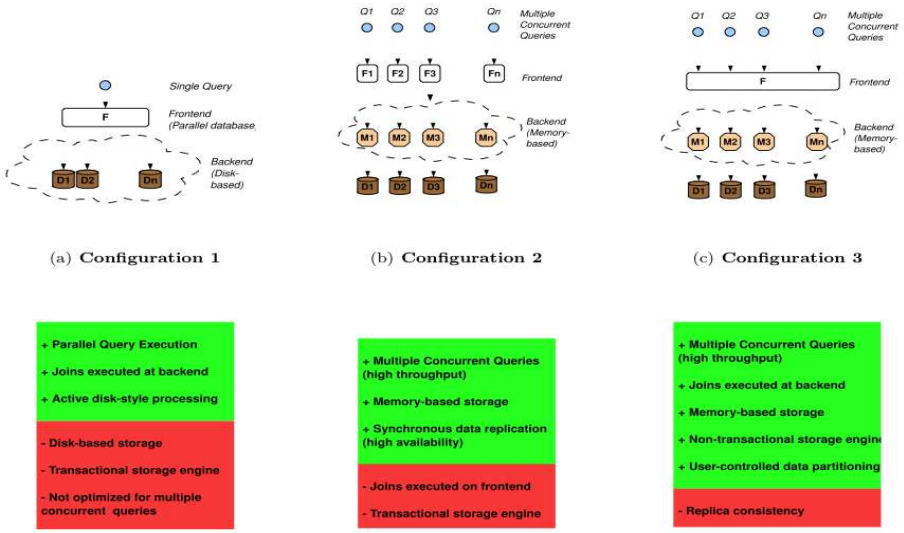

Figure 4. Alternative Database Architecture Configurations, their pros and cons

FOV Region density	# Objects	Approx. # DIASources	# resulting matches
High	*3044468*	*30551*	*53938*
Average	*373763*	*3709*	*4888*
Low	*76073*	*764*	*942*

Table 1. Test Region Characteristics

6.2 Configuration 1: Netezza Performance Server

The `NPS` [2] system at the Lawrence Livermore National Laboratory was used to evaluate configuration 1 described in section 5.1. The frontend of the system is connected via Gigabit Ethernet switch to 56 SPUs. Each SPU had 320 GB local disk with a read bandwidth of 60 MB/sec per SPU disk. The catalog data was distributed uniformly amongst the SPUs based on a hash of the object ID column. The prepare phase on `NPS` extracts objects for the FOV region from the `Object` catalog and loads it into a disk-resident `FOVObject` table distributed across all SPUs.

We made the following changes to the crossmatch algorithms to deploy them on `NPS`: (1) the original schema was modified so that all tables are no longer indexed; the *Zones* and *OptZones* algorithm now work on non-indexed disk-based tables. (2) The query interface for `NPS`, *nzsql* does not support stored procedures and hence, we cannot express loop traversal in *nzsql*. For *OptZones*, we wrote a script outside of the database that emulates the algorithms loop traversal from minimum to maximum zone IDs within an FOV. This script effectively issues a join operation (between the contents of zone i and $\texttt{SecondaryZone}_i$) on behalf of the stored procedure for each of the n zones within an FOV. These joins are then executed as a batch of tasks. (3) The script also inserts `BEGIN TRANSACTION` and `COMMIT` statements at appropriate positions within this batch of tasks in order to minimize transactional overheads that arise from executing each join explicitly as a separate statement. (4) *nzsql* does not support stored functions. Therefore, we developed a a user-defined function (UDF) in C++ to determine the right ascension search range (Δra) for a given zone. The UDF was registered with the `NPS` system and executes on each SPU when the script for *OptZones* is invoked.

	`FOVObject` vs. `DIASource`		
Test FOV Region	**Naive**	***Zones***	***OptZones***
High density	*1950 s*	*51 s*	*210 s*
Average density	*26 s*	*3 s*	*96 s*
Low density	*2 s*	*0.7 s*	*67 s*
	`DIASource` vs. `FOVObject`		
Test FOV Region	**Naive**	***Zones***	***OptZones***
High density	*2739 s*	*52 s*	*280 s*
Average density	*37 s*	*3 s*	*132 s*
Low density	*2 s*	*0.7 s*	*118 s*

Table 2. Crossmatch time on Netezza Performance Server

Table 2 shows the execution time for crossmatch on

NPS assuming that the FOVObject and DIASource are already resident on the SPU disks. We observe that the *Zones* algorithm outperforms other algorithms for all test regions in spite of the fact that it involves a pair of expensive join operations performed without index support on the NPS. This shows that for configuration 1: (1) the query engine of the parallel database system can very efficiently break down even complex queries like the *Zones* algorithm into snippets for parallel execution on the SPUs, and (2) the active disk style processing (FPGAs in the NPS aided by zone-maps) are able to take on the bulk of the data filtering so that the joins can be performed efficiently on the SPUs even without indexes.

The *OptZones* algorithm performs poorly on the NPS as compared to *Zones*. In fact, for the medium and low density regions, even a naive approach does better than *OptZones*. This result defies the theory that the *OptZones* query will always perform better than *Zones* and points to a potential mismatch between the *OptZones* algorithm and this configuration. The following discussion explains this mismatch in greater detail:

- The *OptZones* needs to create and populate on-demand the $\texttt{SecondaryZone}_i$ table prior to performing the join operation between objects within a zone i. This will work well only in configurations where table data is maintained in memory, because data movement from source to destination tables is achieved via either a local in-memory data copy or accessing remote memories over a fast network. However, in this case, the same operation involves a extraction of data from disk on all SPUs, creation of a new on-disk $\texttt{SecondaryZone}_i$ table, and writing the contents of this table onto disk. Suppose an FOV has n zones, then the additional overhead of creating and populating this on-disk table n times factors heavily into the performance of *OptZones* (The zone height and FOV dimensions in our experiments resulted in around 210 zones per FOV).

- One can argue that the $\texttt{SecondaryZone}_i$ tables could also be created beforehand as part of the prepare phase. However, the lack of stored procedures in *nzsql* means that the SQL statement from figure 3 cannot be prepared just once and reified for each zone i. Each of the n queries are instead explicitly issued via an external script as a batch of tasks, thereby entailing query parsing and transactional overheads. We argue that for the *Zones* algorithm, the query style involves a single complex query accessing disk once for large data and hence, is more suited to such a configuration than *OptZones*, where a large number of simpler queries repeatedly access disk for small data.

With sufficient number of SPUs, we believe that the *Zones* algorithm running on NPS will meet LSST constraints. One main concern with this configuration is the high prepare phase time (around 85 seconds per FOV) which places it way outside LSST time constraints. We tried to avoid the prepare phase altogether by crossmatching the on-disk DIASource directly against the entire Object catalog, an operation that took 18 minutes [2]. So, we partitioned the Object catalog into a set of N coarse disjoint chunks C_1, C_2, ..., C_N. Each chunk C_i is represented in the database as a sub-table T_i, whose size is much smaller than the original table. The rows of each sub-table are distributed across all SPUs. So, given an FOV region, we now determine the m chunks that intersect this FOV region using an externally maintained chunk index. The crossmatch for the FOV can then be carried out by simply merging the results from the m crossmatches between DIASource and the sub-table T_i corresponding to intersecting chunk C_i. In this "partition & crossmatch" strategy, we used chunks of size 4.5 degrees × 4.5 degrees, distributed among 48 SPUs (8 SPUs were down at the time of testing). The high-density FOV region intersected with 4 such chunks. The results of the crossmatch against each of these four chunk tables is shown in Table 3 and compared against the time it would take to directly crossmatch the DIASource against a single FOVObject table created during the prepare phase.

DIASource vs.	**Crossmatch time**	**# matching entries**
Chunk 1	*21 s*	*13196*
Chunk 2	*20 s*	*14424*
Chunk 3	*19 s*	*14140*
Chunk 4	*17 s*	*12178*
Total	*77s*	*53938*

Table 3. Crossmatch time on Netezza Performance Server using "Partition and Crossmatch" strategy (*Zones* algorithm used for crossmatch of DIASource against each chunk)

The results show that the crossmatch time for the high density FOV (obtained by adding the crossmatch times for each chunk within the FOV) is 77 seconds. In contrast, the direct crossmatch between FOVObject

[2]Crossmatch time increases exponentially with the size of the tables involved as larger tables imply greater data exchange among the SPUs

and DIASource for the high density FOV took 71 seconds on the same number of SPUs. This shows that "partition & crossmatch" strategy performs only slightly worse as compared to the direct crossmatch between DIASource and FOVObject for the high density FOV on 48 SPUs. Of greater significance is the fact that we no longer need a prepare phase to extract and prefetch data. The increase in crossmatch time, albeit not by much, is expected because, in the absence of index support, detections are compared against every object within each intersecting chunk. To further reduce crossmatch time, we tried to issue these m crossmatch queries concurrently from the NPS frontend. However, NPS and other representative systems for this configuration are not optimized for throughput of execution of multiple concurrent queries, i.e., the hardware accelerators can only service one snippet at a time. Hence, we were restricted here by the sequential nature in which multiple queries are executed by the NPS frontend.

6.3 Configuration 2: MySQL Cluster

This configuration was implemented using a MySQL Cluster instance running on an NSF-funded cluster at the Ohio State University. The cluster consists of 16 AMD dual-processor Opteron-250 nodes, each with 8 GB of memory, giving us a collective memory pool of 128 GB. The nodes are interconnected by both an Infiniband and 1Gbps Ethernet network. The maximum disk bandwidth per node was around 35 MB/sec and 55 MB/sec respectively for sequential reads and writes. The MySQL Cluster instance consists of one manager node, 8 data nodes and upto 8 frontend nodes. A single copy of the data is distributed uniformly among the data nodes.

Table 4 shows execution times for the crossmatch algorithms for the high and average density regions. In these experiments, the crossmatch queries are submitted to a single frontend node. The performance for *Zones* and *OptZones* algorithms are worse than in the previous configuration. This is because, in general, there is a mismatch between the requirements of both the algorithms and this configuration. The core operation in both crossmatch algorithms is the JOIN operation. However, in this configuration, joins are not executed in parallel. Instead, the relevant data from the tables being joined is sent to a single node where the join takes place. The poor performance of *Zones* is attributed to the inability of the MySQL query engine to recognize and use the appropriate index for the JOIN even in the presence of user-provided hints. We also observed that joins and other complex queries in MySQL Cluster are always performed on the frontend node

	FOVObject vs. DIASource		
FOV Region	**Naive**	***Zones***	***OptZones***
High density	*11h 22m*	*11h 58m*	*13m 34s*
Average density	*10m 16s*	*10m 57s*	*18s*
	DIASource vs. FOVObject		
FOV Region	**Naive**	***Zones***	***OptZones***
High density	*9h 27m*	*10h 2m*	*19m 40s*
Average density	*8m 37s*	*9m 15s*	*2m 30s*

Table 4. Crossmatch time on MySQL Cluster

where the query was submitted. The *OptZones* algorithm issues n join queries, one for each of the n zones in an FOV. For each of these queries, the tables referenced by the join operation are transferred over the network from the remote memories on the data node to the frontend node. Moreover, MySQL Cluster does not employ data caching on the frontend nodes. Consequently, even though one of the tables being joined is common to all n queries, it is never cached at the frontend node and is repeatedly transferred over the network n times. This inefficient join mechanism employed in MySQL Cluster causes tremedous data transfer overheads and explains the poor performance of the *OptZones* algorithm in this configuration. Improvements to the join mechansims in future versions of MySQL Cluster will help prevent this mismatch.

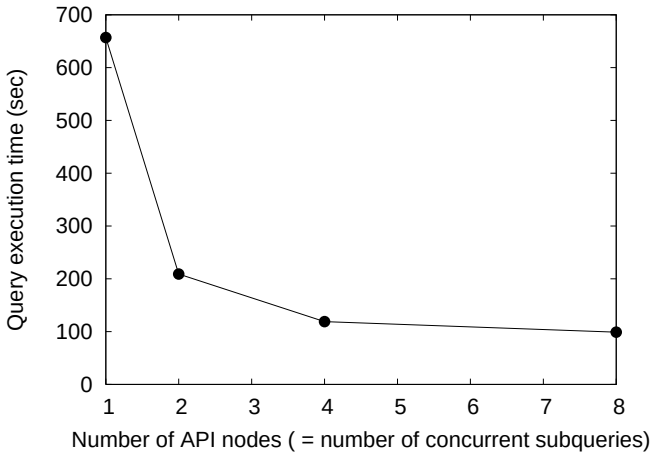

Figure 5. Scalability with number of frontend nodes (8 data nodes and the average density region were used)

The main motivation behind using this configuration was its ability to issue a large number of concurrent subqueries from multiple frontend nodes. Hence, we seek to divide the crossmatch queries into a number of sub-queries that can be executed in parallel. At

first, it might appear that the *OptZones* is better suited to this approach as it implicitly issues multiple join queries (one per zone in the FOV) that can be executed independently provided all the `SecondaryZone`$_i$ tables have been created beforehand. However, the number of zones, n in an FOV is generally much larger than M, the maximum number of permissible frontend nodes in a `MySQL Cluster` instance. As a result, each frontend node would submit (n/M) sub-queries, implying that data is transferred (n/M) times from backend to frontend. The resulting overheads will undo the benefits of concurrent query execution. The ideal scenario is one where data is transferred exactly once to each frontend. This can be achieved by dividing the query into as many concurrent sub-queries as there are frontend nodes. We divided the *Zones* query explicitly into a maximum of M sub-queries, where M is the number of frontend nodes in our MySQL instance. Each frontend node would submit a query responsible for $(1/\mathrm{M})^{th}$ of the overall query workload. Figure 6.3 shows how crossmatch time decreases as we increase the degree of parallelism, i.e., the number of frontend nodes that concurrently execute sub-queries. For the average density FOV, increasing the number of frontend nodes beyond 4 did not improve the crossmatch times. The execution time, even when 8 frontend nodes are used, does not meet LSST requirements. This is because each sub-query of *Zones* also suffers from `MySQL` query engine's non-recognition of appropriate indexes. Assuming other representive systems for this configuration utilize the correct indexes, we argue that this configuration can scale well to a large number of frontend nodes and meet the LSST time constraints.

6.4 Configuration 3: Independent Databases with Replication

This configuration was developed using the same hardware specified in Section 6.3. Each of the 16 nodes ran an independent `MySQL` server with table data stored on local memory. An additional node served as a master node and coordinated the query execution. This hybrid set up can be divided into independent groups of nodes based upon a replication factor: A set of n nodes can be divided into g groups of p nodes each. Data from the entire object catalog is distributed among the nodes in each group. Here, g is the replication factor because copies of data exist in each group, and there are g such groups. At one extreme, we may have an $n \times 1$ grouping, i.e., a single group consisting of n nodes. This corresponds to the case where the data is uniformly partitioned amongst the nodes without any replication (similar to the data distribution schemes in `NPS` and `MySQL Cluster`). At the other extreme, we can have a $1 \times n$ grouping meaning we have n groups of one node each, where data is replicated across all the nodes, and a query can be executed indepenedently by any node/group. By modifying g and p for a given n, we can evaluate intermediary grouping schemes which help us control the data partitioning and replication more flexibly than in our earlier configurations.

We evaluated the *OptZones* algorithm which had to be modified to run in this configuration. The naive algorithm was not evaluated under this configuration because it requires an all-to-all comparison between objects and detections, while this configuration seeks to localize all crossmatch computations and support no communication between the database instances. Replication of the historical data amongst the nodes is done as an offline operation and is not part of our evaluation. Given a FOV, the master node first communicates the FOV to the worker nodes so that they can prefetch the historical objects within that FOV into an in-memory `FOVObject` table (i.e., the prepare phase). The master node then sends the new detections(DIASources) for that FOV to the data nodes. Once the data nodes receive these detections, they perform the crossmatch in memory against only those objects that they locally own. In this confguration, our main goal is to be able to avoid the need for a distributed join operation. Since we are using an independent set of database instances, we need to structure the execution of the crossmatch queries such that the join operations performed on each data node need access to local data only.

The overall query execution time in this configuration includes the time to transfer the detections over the network and the time to execute the query. The amount of data transferred and the computational workload on each data node will depend upon the grouping strategy adopted. In the $n \times 1$ (only partitioning, no replication) case, every join operation potentially needs data from all nodes on account of the uniform partitioning of the Object table. Here, the master node needs to send the DIASources to all the data nodes. This could prove to be a bottleneck in the case where the number of DIASources is extremely large and comparable to the number of objects in the FOV, or in the case where our configuration has been deployed on a cluster with slow network connections between the nodes. This scheme would work well in the case where we have low-power processors connected via high speed communications medium. On the other hand, in the $1 \times n$ case, each join query can be executed by any one of the nodes. So, the master node divides the query workload equally amongst the data nodes and sends only (1/n)th the number of DIASources to each data

node. In this case, we are reducing the volume of data communicated. But each data node will have to crossmatch its share of the DIASource against all objects in the FOV. This case would ideally suit clusters with very fast processors and slower networks. In the intermediate grouping strategies, the master node would need to send a subset of the DIASources to a group of data nodes.

FOV	Prepare time	DIASource transfer time	Query time	Total time
High	*1.9 s*	*5.6 s*	*1.3 s*	*7 s*
Average	*0.9 s*	*5 s*	*0.2 s*	*5.4 s*
Low	*0.8 s*	*5 s*	*0.1 s*	*5.3 s*

Table 5. Crossmatch time using 16x1 strategy (*OptZones* algorithm), `FOVObject` vs. `DIASources`

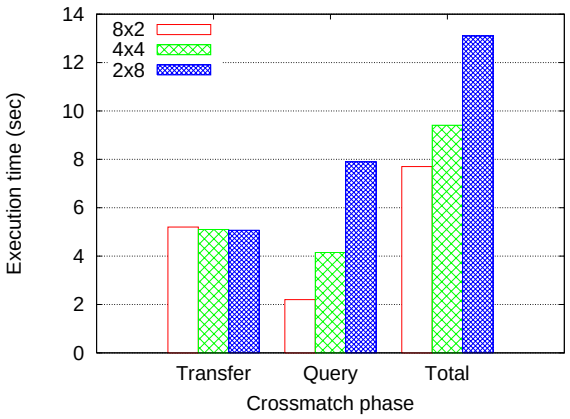

Figure 6. Varying replication factors and query partitioning mechanisms (*OptZones* algorithm, high density FOV region, `DIASource` vs. `FOVObject`)

Table 5 shows the execution times for each phase of the crossmatch in this configuration when we choose to simply partition the data without any replication. The "total" column is the sum of DIASource transfer time, the time to load the DIASources into memory on the data nodes and the query execution time and is measured as the execution time as perceived by the master. The overall crossmatch times obtained using this configuration, especially for the high-density FOV region comfortably outperform the other two configurations. This is because this hybrid configuration combines the best features of the other two configurations. Like configuration 2, it stores all data in the memories of each node, supports indexing of the data and the concurrent execution of multiple sub-queries. Like configuration 1, it performs joins in parallel at the backend nodes (depending on the replication factor for the data). By combining the best features, this configuration is able to achieve superior crossmatch times over other configurations, provided the crossmatch query can be suitably broken down into smaller independent sub-queries. We also note that the performance numbers obtained for this configuration are not specifically tied to `MySQL` database system instances. We believe that the trends observed in these results would be similar when other database engines are used for this configuration in place of `MySQL`.

Figure 6 shows that for the high-density FOV region, the total crossmatch time using this configuration was always within the LSST time constraints. We also investigated the effects of data replication using this configuration. The figure shows that, as the replication factor is increased, the data nodes spend more time on the prepare phase(not shown) and more time on query execution. The transfer time decreases, although only marginally for our test case. The increase in query execution time is explained as follows: As replication factor increases, the data nodes are responsible for execution of smaller parts of the query. That is, they handle the crossmatch of a smaller set of detections. At the same time, the smaller set of detections needs to be crossmatched against a larger number of FOV objects. Since the number of FOV objects is much larger than detections in our use cases, we can expect this trend for most FOV regions. However, in cases where the number of detections and objects in an FOV are comparable and large, we should observe a decrease in transfer times with increasing replication factors and a more gradual increase in the query execution times. In this way, the replication factor can be tuned to extract optimal performance for different sizes of `DIASource` and `FOVObject`.

7 Summary and Conclusions

In this work, we have explored and evaluated database-based solutions to spatial object association or crossmatch, an important spatial data analysis operation that finds use in diverse application domains ranging from astronomy to GIS. We have investigated two variations of spatial crossmatch algorithms, *Zones* [8] and *OptZones* [5]. These two algorithms implement different query styles and optimizations. Our experimental evaluation shows that (1) The *Zones* algorithm performs better than the *OptZones* algorithm on the first database system configuration, because (a) *OptZones* constructs and populates `SecondaryZone`

tables on-demand for every zone in an FOV. These tables are created on disk (as opposed to memory) and distributed across all the nodes in the backend. The overhead of creating disk-based tables on-demand for each zone leads to poor performance; (b) The lack of support for stored procedures in the system resulted in overheads due to external scripts being treated as independent queries. (2) On the second database system configuration, the algorithms do not perform as well as they do on the first configuration. This is because the second configuration does not execute a query in parallel and executes `JOIN`s on the frontend. However, performance of the algorithms can be improved by partitioning a query into a set of smaller queries and executing these sub-queries as a batch, to take advantage of the high-throughput oriented design of the second configuration. (3) The third configuration enables different partitioning and replication of the catalog tables across a collection of the independent database system instances. The performance results obtained from the *OptZones* algorithm on this configuration indicates that the query execution time increases as the amount of data replication increases. This is because replication reduces the amount of `DIASource` entries broadcast to multiple nodes, but increases the time for the prepare and query phases, since each node has to deal with a larger portion of the `FOVObject` table. Even in our extreme case, the size of the `DIASource` was relatively small. It can be expected that if `DIASource` is much larger, then a configuration that supports replication of portions of the `FOVObject` table could be more efficient, since replication will reduce the amount of `DIASource` entries broadcast.

Acknowledgements. The authors wish to thank the late Marcus Miller for his help in configuring the Netezza systems at Lawrence Livermore National Laboratory (LLNL) and Serge Monkewitz of the Infrared Processing and Analysis Center (IPAC) for providing access to relevant algorithms and datasets. The authors also wish to thank Sergei Nikolaev and Don Dossa of LLNL for their help in understanding LSST application details.

References

[1] LSST – Large Synoptic Survey Telescope. `http://www.lsst.org/`.

[2] The Netezza FAST Engines Framework: A Powerful Framework for High-Performance Analytics. `http://www.netezza.com/documents/whitepapers/fastengines.pdf`. Netezza Technical White Paper, 2007.

[3] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 81–91, New York, NY, USA, 1998. ACM.

[4] J. Becla, A. Hanushevsky, S. Nikolaev, G. Abdulla, A. S. Szalay, M. A. Nieto-Santisteban, A. Thakar, and J. Gray. Designing a multi-petabyte database for LSST. *The ACM Computing Research Repository (CoRR)*, abs/cs/0604112, Apr 2006.

[5] J. Becla, K.-T. Lim, S. Monkewitz, M. Nieto-Santisteban, and A. Thakar. Organizing the extremely large LSST database for real-time astronomical processing. Sep 2007. 17th Annual Astronomical Data Analysis Software and Systems Conference (ADASS 2007), London, England.

[6] M. Beynon, R. Ferreira, T. M. Kurc, A. Sussman, and J. H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, pages 119–134, 2000.

[7] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[8] J. Gray, M. A. Nieto-Santisteban, and A. S. Szalay. The zones algorithm for finding points-near-a-point or cross-matching spatial datasets. *The ACM Computing Research Repository (CoRR)*, abs/cs/0701171, Jan 2007.

[9] M. A. Nieto-Santisteban, A. R. Thakar, and A. S. Szalay. Cross-matching of very large datasets. In *National Science and Technology Council(NSTC) NASA Conference*, 2007.

[10] S. Papadomanolakis, A. Ailamaki, J. C. Lopez, T. Tu, D. R. O'Hallaron, and G. Heber. Efficient query processing on unstructured tetrahedral meshes. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 551–562, New York, NY, USA, 2006. ACM.

[11] R. A. Power. Large Catalogue Query Performance in Relational Databases. *Publications of the Astronomical Society of Australia (PASA)*, 24:13–20, May 2007.

[12] M. Ronstrm and L. Thalmann. MySQL Cluster Architecture Overview, High Availability features of MySQL Cluster. MySQL Technical White Paper, 2004.

[13] A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar. Indexing the sphere with the hierarchical triangular mesh. Technical Report MSR-TR-2005-123, Microsoft Research, Aug 2005.

[14] M. Taylor. Crossmatching developments. *DS3 Report, VOTech Stage 6 Planning Meeting*, 2007.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.